SystemVerilogシミュレーション

活用チュートリアル 02

第2回 テストベンチの拡張

森田栄一

デバイスの記事

本連載では，市販シミュレータの評価版を操作しながら，SystemVerilogを利用した検証手法について学習していく．今回はランダム・パターンを利用したテストベンチの作成法やカバレッジ計測について解説する．シミュレータとしては，米国Mentor Graphics社の「Questa」を使用する．本記事で使用するシミュレータの評価版は，本誌2006年4月号の付属CD-ROMに収録されている．（編集部）

前回（第1回，本誌2006年4月号，pp.122-132）はSystem Verilogを用いたシミュレーション環境を体験するために，シミュレータの基本的な使用方法を紹介しました．また，第1回の最後の演習では，既存のテストベンチを編集して，検証対象に対するさまざまな入力を試してみました．

● 今回はテストベンチの拡張がテーマ

テストベンチに最低限必要な機能は，検証対象に対して信号パターンを与えることです．これに，出力された結果の自動比較機能やパターン自動発生機能を追加することで，より効率的なデバッグを行えるようになります．

一般的なテストベンチのパターン発生方法は，HDL動作記述によるものと，シミュレータ固有のパターン生成機能を用いるものの二つに分けられます．本連載で使用しているシミュレータもパターン生成機能を持っていますが，シミュレータ独自の方法ですべてのテストベンチを作成することはあまり好ましくありません．HDL記述の場合は，それが標準言語であるために，ほかのツールに持ち込んだときにも「互換性」を保つことができます．一方，シミュレータに依存したパターンでは，汎用性が低くなってしまいます．

SystemVerilogは，HDL記述ベースのテストベンチに大きな機動力を与えてくれます．今回は次のようなステップでチュートリアルを進めていきます．

- ランダム・パターン発生機能の追加
- カバレッジ計測機能の追加
- 複数シミュレーションでのカバレッジ向上

● パターンの自動発生とは

パターンの自動発生というのは，ある一定のルールを設定しておいて，一つ一つソースを記述することなくパターンを発生してしまおう，という考えかたです．サンプルの設計で考えると，**リスト1**のようなループ文を用いて，いくつかのメモリ・アドレスへのアクセスを記述することも簡単な自動発生の方法になります．この方法では，ループの回数を多くすれば大量のパターン発生が可能ですが，アドレスを順番に指定しながらのアクセスはあまり現実的な検証とはいえません．

ここで登場するのが，ランダム・パターンという考えかたです．疑似乱数のアルゴリズムを用いて，入力する値を

リスト1　単純なループ記述

```
// 50個のアドレスへの書き込み
for (a = 0; a < 50; a++)
write(a, a);

// 書き込んだ結果を読み出し
for (a = 0; a < 50; a++)
read(a);
```

KeyWord　SystemVerilog，ランダム・パターン，テストベンチ，randomize()メソッド，Questa，シード，ソース・コード・カバレッジ，ファンクショナル・カバレッジ，クロス・カバレッジ，バッチ・モード

ランダムに発生したり，プログラム・シーケンスをランダム化したりします．SystemVerilogは，ランダム・パターンを発生するためのさまざまな機能を備えています．その中の一部を用いて，実際にランダム・パターン発生機能を組み込んだソース・コードを紹介します．

● 新しいソース・コードをダウンロード

筆者の所属する会社のWebサイト（http://www.mentorg.co.jp/dw/）に，今回利用する新しい「example」ディレクトリを準備しました．ページ内に記述された内容に従ってダウンロードしてください．また，同ページ内にインターネットからの必要なソフトウェアの入手方法についても掲載しています．本誌2006年4月号の付属CD-ROMをお持ちでない場合には，こちらをご利用ください．ダウンロードして展開した後，前回と同じように「C:\example」にコピーしてください．前回のデータは，ディレクトリ名を変えてバックアップをとるなどして，新しいデータとすべて入れ替えてください．データに含まれるデザインのアルゴリズムや機能・構成はまったく同じですが，演習の便宜上，若干の変更を加えてあります．また，テストベンチ拡張のための記述も追加しています．

● ランダム・パターン発生記述を書く

ソース・コードproc.svを参照してください．**リスト2**の記述が追加されています．本連載では，SystemVerilogの文法については最低限の解説しか行いません．言語の文法や記述方法の詳細については，各種参考書などを参照してください．

リスト2　ランダム・パターン発生記述

```
/**************** ランダム定義 START ****************/
class rand_sig;
    rand bit [7:0] a;
    rand bit [15:0] d;
    constraint d_range {
        solve a before d;
        (a < 128) -> d inside {[0:9999]};
        (a >= 128) -> d inside {[10000:19999]};}
endclass

rand_sig rand_i = new;

int rep;
       :
    <中略>
       :
/**************** ランダム・パターン発生 START ****************/
//  rand_i.srandom(100);
repeat (100) begin

    randsequence (main)
    main: IDLE RDWR;
    RDWR: READ := 1| WRITE := 5;

    READ: {
        randcase
            1: rep = 3; 2: rep = 2; 3: rep = 1;
        endcase
        $display ("-- READ  x %3d times", rep);
        state = READ;
        repeat (rep) begin
            if (rand_i.randomize() == 1)
                read(rand_i.a);
            else
                $display ("Random Failed");
        end
        };
    WRITE: {
       :
    <中略>
       :
        };
    IDLE: {
       :
    <中略>
       :
        };
    endsequence
end
```

ランダム変数をクラスで定義
制御条件
クラス・インスタンス
繰り返し変数の定義
srandom()メソッド
IDLE→RDWRの順番で実行
RDWRではREADとWRITEが1：5の確率で発生
READ内の処理：randcaseによって繰り返し回数repを確定
repが3，2，1となる確率はそれぞれ1/6，1/3，1/2
全体を100回繰り返し
ランダム・シーケンス
WRITE，IDLEの処理もREADと同様

ランダム・パターン発生の構成は，以下のとおりです．

● クラスrand_sig内に8ビットのaと16ビットのdを宣言

d_rangeという制約条件を定義しています．a(アドレス)の値が128より小さい場合，d(データ)は0～9999の値を，128以上の場合，dは10000～19999の値を発生します．solve行で，aの値が先に確定するように指定しています．なお，この制約は制約条件の動作を確認するのが目的です．検証対象の動作に対して特別な意味は持ちません．

● rand_iという名まえでクラスをインスタンス

クラス定義だけでは，オブジェクトは生成されません．かならずインスタンスを作成する必要があります．

● randsequenceを用いた実行フロー制御

基本サイクルは，IDLE → RDWRが実行されます．IDLEでは，randcaseを用いて繰り返し変数repの値を確定しています．randcaseの中の1: rep = 10; 2: rep = 5; 3: rep = 3;という記述によって，それぞれのrepへの代入文の発生確率を1：2：3に指定しています．RDWRのサイクルでは，READとWRITEが1：5の確率で実行されます．READ/WRITEのサイクル内では，randcaseによって発生したrepの回数分，readとwriteのタスクを実行します．

● randomize()メソッドを用いた乱数の発生

クラス・インスタンス名に続いて.randomize()を記述することで，クラス内の変数のランダム生成が実行されます．このようなクラス・インスタンスに対する実行を，「randomize()メソッドの実行」と呼びます．ランダム・パターン発生のためにこのようなメソッドがいくつか準備されています．

● read/writeタスクへの乱数の代入

発生したaとdの値をread/writeタスクの引き数として実行しています．

● repeatによるrandsequenceの繰り返し

上記のフローをrepeat文によって100回繰り返すようになっています．

また，randsequenceの実行中に，IDLE，READ，WRITEという状態をSystemVerilogの列挙型変数stateへ出力しています．これは，実行されているタスクを波形モニタ上でわかりやすくするためのくふうで，検証対象の動作とは関係ありません．テストベンチには，しばしばこのような検証対象とは直接関係のない記述を追加して，検証内容の確認を行うことがあります．とくにランダム・パターンを用いる場合には，このような手法は有効になります．

● ランダム・パターンによるシミュレーションを実行

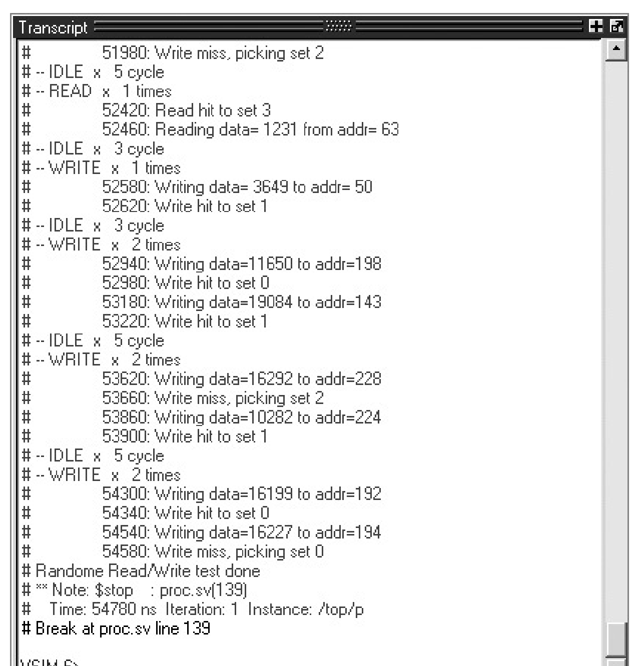
実際にシミュレーションを流してみましょう．Questaを起動して，exampleディレクトリに移動します．コマンドラインから次のコマンドを入力します．

```
QuestaSim> vlib work
QuestaSim> vlog *.sv
QuestaSim> vsim top -wlf first.wlf
```

シミュレーション・セッションが起動します．同時に，今回のセッションの波形をfirst.wlfというログ・ファイルに保存します．次に，Waveウィンドウへ観測信号を追加して，シミュレーションを実行します．

```
VSIM> add wave /*
VSIM> add wave /p/state
VSIM> run -all
```

コマンド・ラインにread/writeタスクの実行状況を示すメッセージが表示されます(**図1**)．今回のデータでは，


Transcript
```
#            51980: Write miss, picking set 2
# -- IDLE  x   5 cycle
# -- READ  x   1 times
#            52420: Read hit to set 3
#            52460: Reading data= 1231 from addr= 63
# -- IDLE  x   3 cycle
# -- WRITE  x   1 times
#            52580: Writing data= 3649 to addr= 50
#            52620: Write hit to set 1
# -- IDLE  x   3 cycle
# -- WRITE  x   2 times
#            52940: Writing data=11650 to addr=198
#            52980: Write hit to set 0
#            53180: Writing data=19084 to addr=143
#            53220: Write hit to set 1
# -- IDLE  x   5 cycle
# -- WRITE  x   2 times
#            53620: Writing data=16292 to addr=228
#            53660: Write miss, picking set 2
#            53860: Writing data=10282 to addr=224
#            53900: Write hit to set 1
# -- IDLE  x   5 cycle
# -- WRITE  x   2 times
#            54300: Writing data=16199 to addr=192
#            54340: Write hit to set 0
#            54540: Writing data=16227 to addr=194
#            54580: Write miss, picking set 0
# Randome Read/Write test done
# ** Note: $stop    : proc.sv(139)
#    Time: 54780 ns  Iteration: 1  Instance: /top/p
# Break at proc.sv line 139

VSIM 6>
```

図1　シミュレーションのメッセージ出力
IDLE，READ，WRITEの各サイクル数と，アクセスしたアドレス，およびデータの値を確認できる．

表示されるaddrとdataの値を両方とも10進数表記にしています．メッセージから次の内容を読み取ります．

- リード・サイクルとライト・サイクルの発生確率
- 各サイクルの繰り返し回数
- アイドルのサイクル数
- 発生したアドレスとデータの値が制約条件を満たしているかどうか

同時に，Waveウィンドウを観測することでrand sequenceによるランダム・シーケンスを確認できます(**図2**)．state変数の値を中心に確認してみてください．

● 同じランダム？ 本当のランダム？

ランダム・パターンによる検証でかならず考慮しなければならないのが，次の2点です．

- 問題が発生したシーケンスを再現する(同じランダム)

ランダム・パターンを実行して，なんらかのエラーが発見された場合，デバッグするためには，問題が発生した状況を再現する必要があります．設計者が思いつかないようなパターンを自動発生してくれるランダム検証だからこそ，「再現」の重要性は非常に大きくなります．

- 毎回異なったパターンを発生する(ほんとうのランダム)

シミュレーションを何度実行してもまったく同じランダム・パターンが実行されるのでは，ほんとうにランダムとはいえません．複数回のシミュレーションで検証の達成率を上げるためには，セッションごとに異なったパターンを発生できなければ意味がありません．

本来の乱数発生では，次に発生する数を予測できません．ところがコンピュータの世界では，数式アルゴリズムによって発生させる「疑似乱数」しか取り扱うことができません．これは，アルゴリズムと条件がわかっている場合，発生する乱数列を計算で求めることができます．このアルゴリズムの中で乱数発生を制御しているのが「シード」と呼ばれる数値です．「同じランダム」を実現するには，同一のシードを用いてシミュレーションを実行します．「ほんとうのランダム」を実行するには，毎回シードを変更します．

SystemVerilogによるシミュレーションのシード変更には2通りの方法があります．一つは，ソース・コード記述中でsrandom()メソッドを用いて変更する方法です(**リスト2**のコメント・アウト部分を参照)．もう一つは，シミュレータの機能でコントロールする方法です．srandom()を用いる方法では，ソース・コードの変更後に，再コンパイルを行う必要があります．またクラス・イ

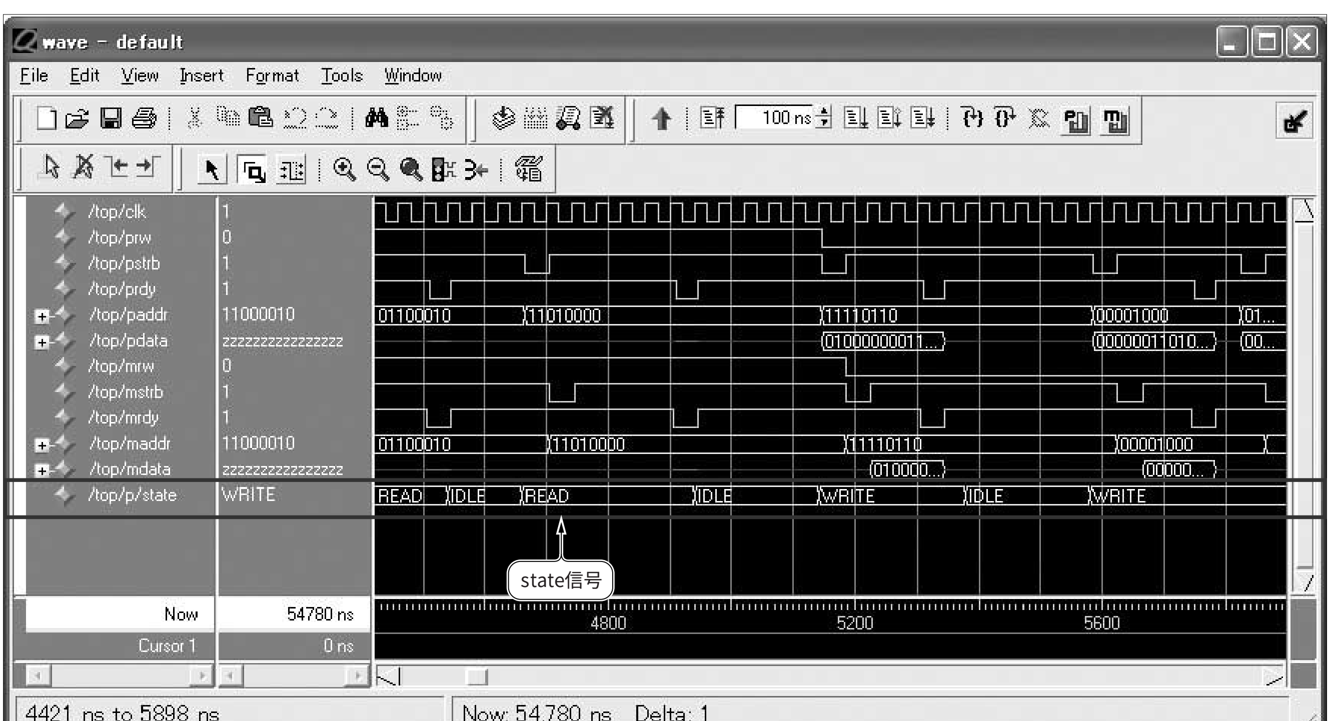


図2　Waveウィンドウによるランダム・シーケンスの確認
ランダム・パターン発生のようすを確認する．state信号に実際のパターン発生のシーケンスが出力される．

ンスタンスに対するメソッドのため，randsequenceやrandcaseに対するシードを変更することはできません．これらのことから，シードはシミュレータの機能でコントロールするほうが便利であるといえます．

本ツールでは起動時に-sv_seedオプションを用いてルート・シードを変更できます．このルート・シードをもとに，一定のルールで値を発生させて，クラス・インスタンスごと，randsequence/randcaseごとにシードを割り振ります．シミュレータのオプションによるシード指定は，srandom()による明示的な指定を上書きしません．つまり，ソース・コード上で同一のランダム・パターン発生を維持したい部分についてのみsrandom()メソッドを用いるのが一般的な方法になります．

● シードを変更してシミュレーション

次にシードを変更してシミュレーションを実行してみます．現在起動しているシミュレーション・セッションをいったん終了します．

```
VSIM> quit -sim
```

シードを指定してシミュレーション・セッションを起動します．

```
QuestaSim> vsim -sv_seed 200 top
```

シミュレーション・セッションの起動後，Waveウィンドウへ信号を追加してシミュレーションを実行します．

```
VSIM> add wave /*
VSIM> add wave /p/state
VSIM> run -all
```

最初のセッションとの違いを確認するために，first.wlfをローディングして並べて比較します．次のコマンドを実行します．

```
VSIM> dataset open first.wlf
```

次にWaveウィンドウをundockします．Waveウィンドウのメニューから「Insert」→「Window Pane」を選択します．Waveウィンドウが二つの領域（ペーンと呼ぶ）に分割されます（**図3**）．この二つの領域はそれぞれが縦スクロール・バーを持っているので，任意の信号を並べて比較する場合に便利です．下側の何も表示されていないペーンをクリックして選択します．選択されたペーンは，左側に白いバーが表示されます（**図3**）．次のコマンドを実行します．

```
VSIM> add wave first:/*
VSIM> add wave first:/p/state
```

このコマンドで，first.wlf内に保存された信号が下側のペーンに追加されます（**図4**）．state信号を中心に確認することでシーケンスの違いを容易に確認できます．同時に，アドレスとデータの値から発生した数値が異なっているこ

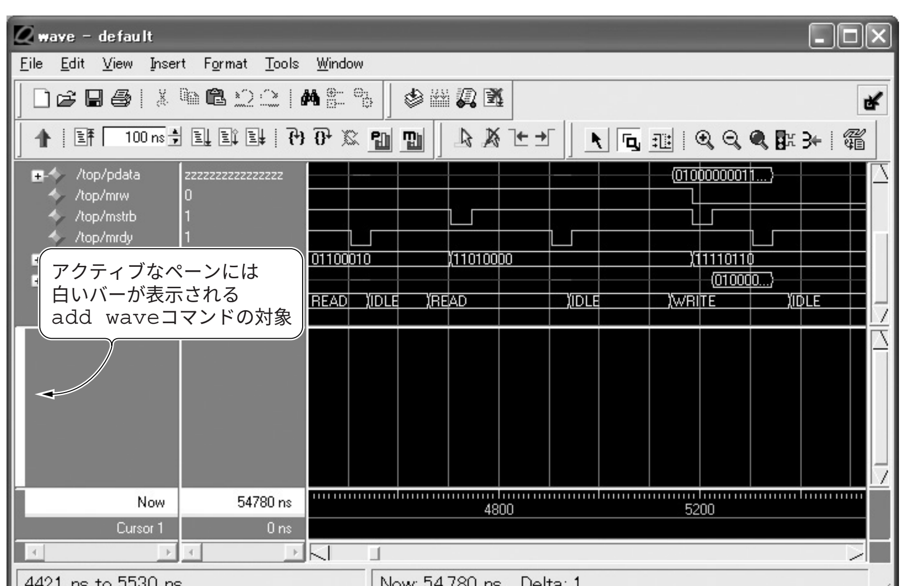


図3
複数のウィンドウ・ペーン
Waveウィンドウを分割することで，信号系列ごとに分けて表示できる．それぞれのペーンをスクロールすることで，特定の信号どうしを並べて比較する．

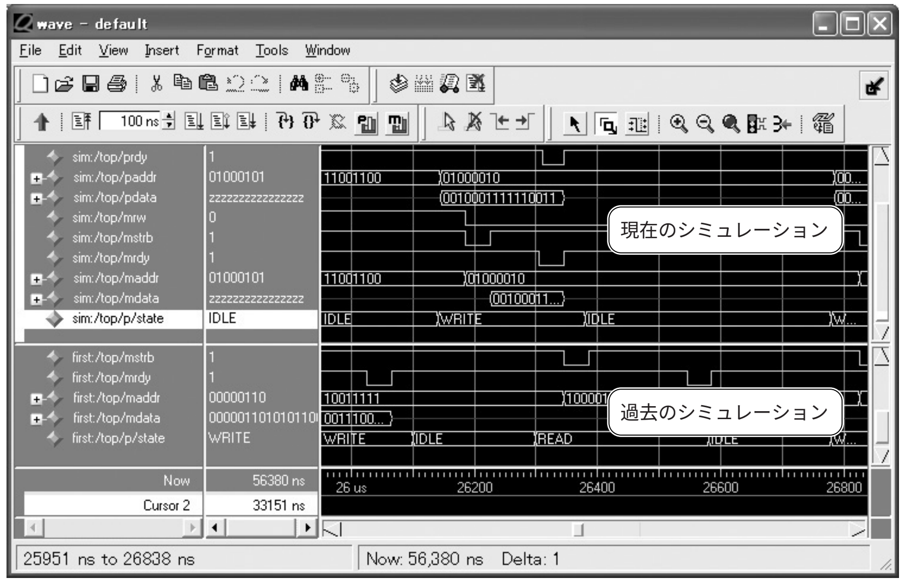


図4
二つのシミュレーション結果の比較
上下のペーンに並べた二つのシミュレーション結果を比較した．発生するサイクルの違いを中心に確認する．

とがわかります．さらに何度かシードを変更して，同じように違いを確認してみてください．

次のステップに進む前に，ソース・コードを編集します．現在の設定では，ライト・サイクルの発生確率が高くなっていますが，proc.svを編集して次のように変更します．

```
現在：RDWR: READ := 1| WRITE := 5;
変更後：RDWR: READ  |  WRITE;
```

それぞれの処理が同一確率の場合は，比率の数字を省略できます．

● カバレッジを計測する

ここまでの実行で，ランダム・パターン生成の便利さが理解できたと思います．ただし，このように自動的に大量のパターンの発生を行った場合，その結果として何が検証できたかを確認する必要があります．このような検証項目の達成度をカバレッジといいます．カバレッジは，ある一定の項目が実行されたかどうかを観測します．テストベンチを流した結果，「実行されていない項目」は「検証されていない項目」と言えます．

カバレッジには，次のような種類があります．

1）ソース・コード・カバレッジ

ソース・コードの内容が実行されたかどうかを確認するもっとも単純なカバレッジです．カバレッジ計測機能をもったシミュレータがあれば，とくに追加の記述を行う必要はありません．本シミュレータもコード・カバレッジ機能を持っています．ソース・コード・カバレッジには，「ステートメント」，「ブランチ」，「コンディション」，「エクスプレッション」，「ステート・マシン」などの種類があります．

2）ファンクショナル・カバレッジ

テストベンチの実行によって，一定の動作（ファンクション）が実行されたかどうかを観測します．検証対象が一定の状態に達したかどうかを確認する「データ指向カバレッジ」と，特定信号パターンのシーケンスが実行されたかどうかを確認する「コントロール指向カバレッジ」の2種類があります．今回は，データ指向カバレッジの計測を行います．

「回路の状態」に対する達成率を計測するには，System Verilogのcovergroupという機能を用いてこれを実現します．実際の記述をベースに解説しましょう．exampleディレクトリのcoverフォルダ内のcache.svで現在のcache.svを上書きします．このソース内には，**リスト3**に示すカバレッジ記述が挿入されています．

● covergroupによるカバレッジ記述

covergroup cache_coverで，カバレッジを計測する対象を定義します．最初の@(posedge clk)は，カバレッジのサンプリング・クロックです．対象の設定には，coverpointを用います．下記のような設定を行っています．

リスト3
カバレッジ計測の記述

```
/**************** カバレッジ定義 START ****************/

covergroup cache_cover @ (posedge clk);

    hit_cov: coverpoint hit iff (!pstrb) {
        bins s0   = {4'b0001};
        bins s1   = {4'b0010};
        bins s2   = {4'b0100};
        bins s3   = {4'b1000};
    }

    miss_cov: coverpoint hit iff (!pstrb) {
        bins miss = {4'b0000};
    }

    rw_cov: coverpoint prw iff (!pstrb) {
        bins write = {0};
        bins read  = {1};
    }

    eaddr_cov: coverpoint paddr[4:0] iff (!pstrb);

    cache_miss: cross eaddr_cov, miss_cov {
//        option.at_least = 5;
//        type_option.weight = 5;
    }

    hit_once: cross  rw_cov, hit_cov, eaddr_cov  {
 //        type_option.weight = 10;
    }

endgroup

cache_cover cov1 = new;

/**************** カバレッジ定義 END ****************/
```

注釈：
- covergroup定義
- クロックの立ち上がりでサンプリング
- coverpoint指定 四つのbinsを指定 キャッシュ・セットごとのヒット回数
- キャッシュ・ミスの回数
- リードとライトの回数
- 各エントリ・アドレス
- クロス・カバレッジ
- covergroupインスタンス

●hit_cov

4ビットの信号hitに対するカバレッジです．hit信号は，各ビットが四つのキャッシュ・セットに接続されています．プロセッサからのアドレス入力に対していずれかのキャッシュ・セットにヒットした場合，特定の1ビットに‘1’が出力されます．hit信号の条件によってbinsを定義しています．binsというのは，ある条件に合致した件数を保存するための箱のようなものです．四つのいずれかのキャッシュ・セットのキャッシュ・ヒット判断によって，四つのbinsを定義しています．

●miss_cov

信号hitに対して，すべてのキャッシュ・セットでヒットなし（つまりキャッシュ・ミス）のbinsを定義しています．

●rw_cov

prw信号の状態を観測することで，プロセッサからの命令がリードなのかライトなのかを観測します．それぞれ，READとWRITEという名まえのbinsを定義しています．

●eaddr_cov

アドレス（paddr）の下位5ビット（キャッシュ・アクセスのエントリ・アドレス）を観測することで，キャッシュの何番目のアドレスにアクセスがあったかを確認します．とくにbinsを定義していないため，自動的に全アドレスに対するbinsが生成されます．すべてのcoverpointにはサンプリング有効条件iff (!pstrb)を付加しています．これは，strb信号のアクティブ時に測定を限定することで，正確なリードとライトの回数を計測するためです（**図5**参照）．crossというキーワードで，二つの「クロ

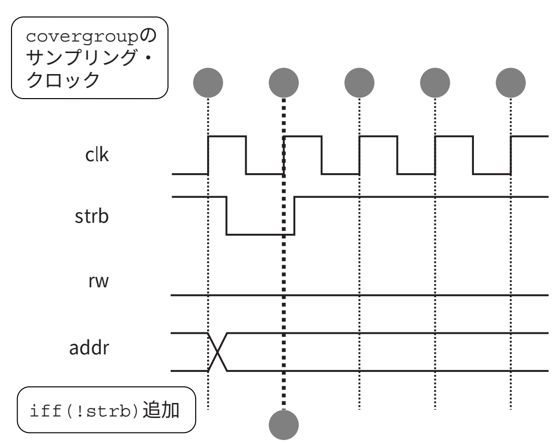


図5　カバレッジ・サンプリング・ポイント（ライト・サイクル時）
全クロックでサンプリングを行うと，一つのライト・サイクル中に5回のサンプリングが行われる．観測したいデータは，strb信号がアクティブのときの値．

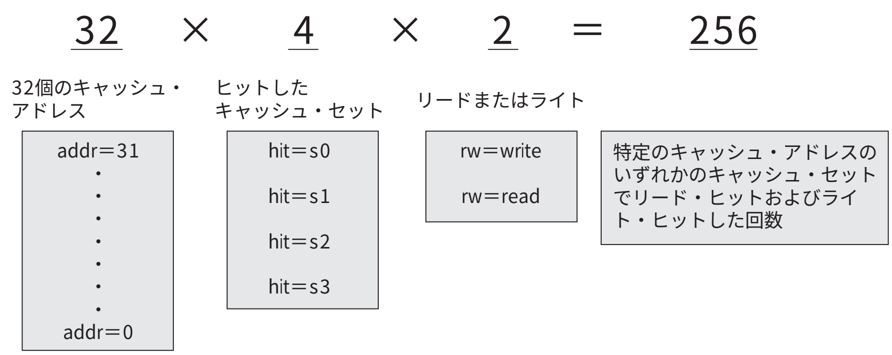


図6
クロス・カバレッジ
複数の`coverpoint`**の状態を組み合わせたマトリクスに対するカバレッジを測定する．複雑な組み合わせ条件に対するカバレッジを簡単に指定できる．**

ス・カバレッジ」を定義しています．クロス・カバレッジは，複数の`coverpoint`の組み合わせ状態を観測します（**図6**）．

● `cache_miss`：

`miss_cov`と`eaddr_cov`のクロス・カバレッジです．取得されるカバレッジの回数は，特定キャッシュ・アドレスに対するキャッシュ・ミスの回数になります．

● `hit_once`：

`hit_cov`と`rw_cov`，`eaddr_cov`のクロス・カバレッジです．特定アドレスの特定キャッシュ・セットに対するリード/ライト・ヒットの回数になります（**図6**）．合計256個の`bins`が生成されます．

`covergroup`定義に続いて，`covergroup`をインスタンスしています．クラスと同じように，定義だけでは実際のシミュレーションで利用できません．また，インスタンスはデザイン・ローディング時ではなく，シミュレーションの初期化のステージで実行されます．つまり，「run」を実行しないと，カバレッジのオブジェクトを確認することはできません．

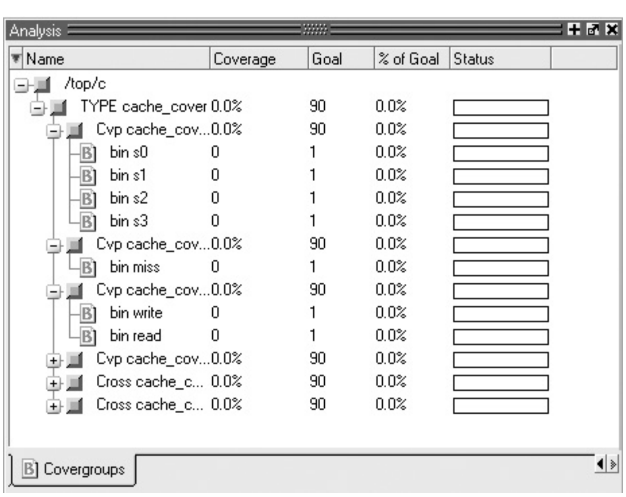


図7　Analysisウィンドウ
Covergroupsタブにカバレッジ・オブジェクトの情報が表示される．シミュレーションが実行されていないため，すべての`bins`**のカウントはゼロ．**

● カバレッジ記述はどこに書く？

カバレッジ記述については，サンプルのようにデザイン内に記述する場合と，テストベンチ内に記述する場合の2通りが考えられます．それぞれ下記の目的があります．

● デザイン内に記述する場合

対象デザインが再利用される際に，どのような階層中に組み込まれた場合もカバレッジ計測を利用できます．もし，複数インスタンスされたとしても，とくに記述を追加することなくインスタンスごとのカバレッジを取得できます．

● テストベンチに記述する場合

設計記述に手を加えずにカバレッジを計測できます．また，システム・レベルのカバレッジで，複数のモジュールに対する横断的なカバレッジを取得したいときは，テストベンチにまとめてカバレッジ記述を行います．この方法では，モジュールの構成に応じてカバレッジ記述内の階層名を書き換える必要があります．複数階層で同一のカバレッジ計測を行うには，パラメータを利用したカバレッジを使用するのが便利です．

● カバレッジを使用してシミュレーション

コンパイル実行後，設計データをローディングします．

```
QuestaSim> vlog *.sv
QuestaSim> vsim top
```

最初にメイン・メニューから，「View」→「Analysis」→

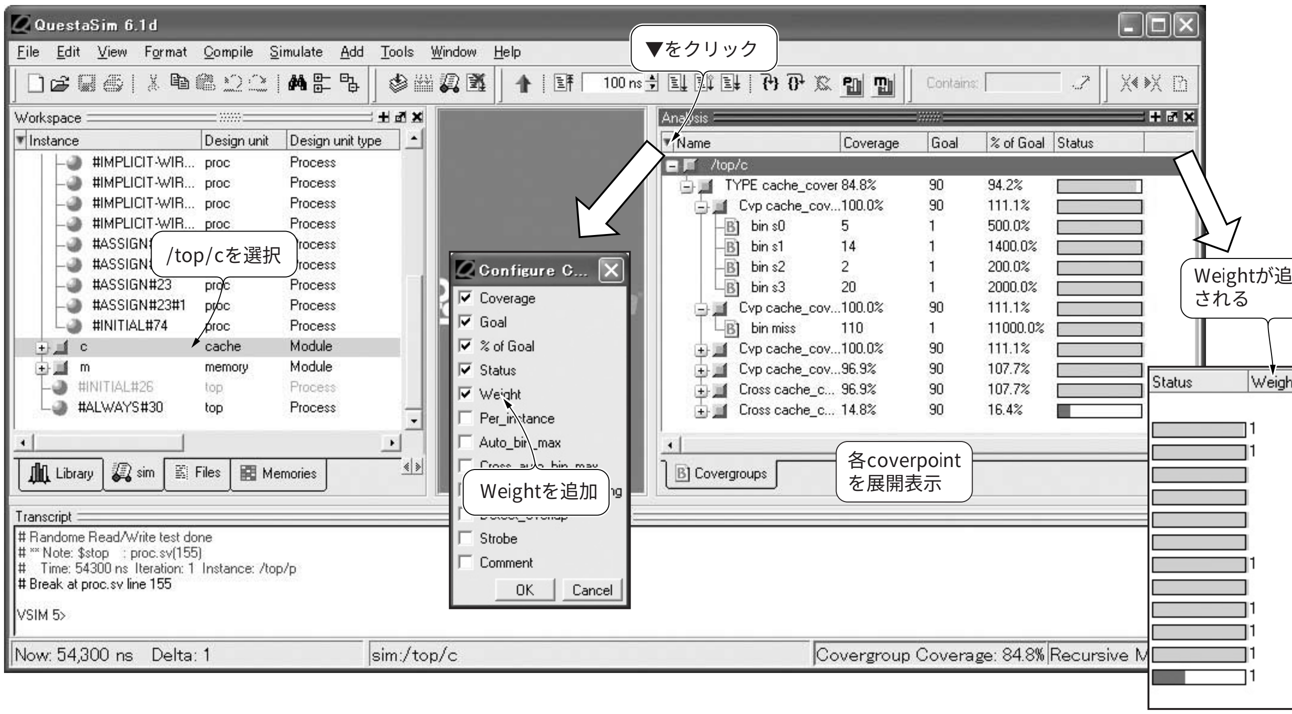


図8　シミュレーション実行後のカバレッジ表示
Workspaceで選択した階層以下に定義されているカバレッジが表示される．カバレッジの表示情報にWeight（重みづけ）項目を追加する．

「Covergroups」を選択します．続いて，次のコマンドを実行します．

```
VSIM> run 0
```

すると，**図7**のAnalysisウィンドウが表示されます．「run 0」でシミュレーションの初期化ステージが実行され，covergroupのインスタンスが生成されます．Analysisウィンドウのcovergroupsタブ中には，各coverpointやクロス・カバレッジのbinsが表示されていることを確認できます．シミュレーションを実行します．

```
VSIM> run -all
```

シミュレーションが終了した時点で，キャッシュの階層/top/cを選択します（**図8**）．Analysisウィンドウに各binsに対するカバレッジが表示されます．**図8**を参考に，カバレッジ表示項目の左にある「▼」をクリックして，Weight項目を追加します．Weightは「重みづけ」を意味します．

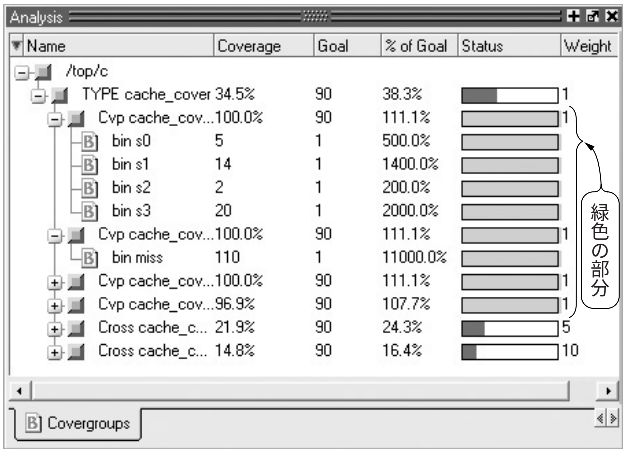
● GOALとWeightを変更

カバレッジ項目内の各binsのGOALは，1（回）に設定されています．coverpointに対するGOALは，90（%）と設定されています．これは，SystemVerilogで規定された初期値になっています．各binsは1回実行されれば「カバーされた」と判断されます．このデフォルト設定によって，全体のカバレッジは90%を超えています（**図8**）．ところが，クロス・カバレッジhit_onceは非常に低いカバレッジしか達成できていません．これは，6個あるcoverpointとクロス・カバレッジのうちの五つがカバーされていることによって，1項目だけカバレッジが低いことが見えなくなっているためです．

本検証のゴールは，次の二つです．

- 全アドレスに対するキャッシュ・ミスが5回以上（キャッシュの掃き出しが行われる）
- 全キャッシュ・セットに対して，リードとライトの両方でキャッシュ・ヒットする

このような検証の目的に合わせて，GOALとWeightを変更します．ソース・コード中（**リスト3**）の「option.at_least...」と「type_option.weight...」のコメント・アウト（3行分）を解除します．このオプションによって，各アドレスに対するキャッシュ・ミスはGOAL（オプション名はat_least）が5回に設定されます．また，Weightが5に設定され，ほかのcoverpointに対して5



Name	Coverage	Goal	% of Goal	Status	Weight
/top/c					
TYPE cache_cover	34.5%	90	38.3%		1
Cvp cache_cov...	100.0%	90	111.1%		1
bin s0	5	1	500.0%		
bin s1	14	1	1400.0%		
bin s2	2	1	200.0%		
bin s3	20	1	2000.0%		
Cvp cache_cov...	100.0%	90	111.1%		1
bin miss	110	1	11000.0%		
Cvp cache_cov...	100.0%	90	111.1%		1
Cvp cache_cov...	96.9%	90	107.7%		1
Cross cache_c...	21.9%	90	24.3%		5
Cross cache_c...	14.8%	90	16.4%		10

図9　カバレッジ条件の変更結果
二つのクロス・カバレッジのWeightが大きくなったことで，全体のカバレッジが低くなっている．

倍の重要度を持ちます．さらに，クロス・カバレッジhit_onceは`Weight`が`10`となり，検証全体への重要度がもっとも高く設定されます．ソース・コードを変更したあと，再シミュレーションを行います．

```
VSIM> vlog -incr *.sv
VSIM> restart -f
VSIM> run -all
```

カバレッジの結果を確認します(**図9**)．先ほどと比べると，cache_missのGOALが上がったことによってカバレッジの緑色の表示が少なくなっています．また，二つのクロス・カバレッジのWeightが大きくなっているため，全体のカバレッジが大幅に減少しています．

このような設定は，カバレッジ計測でもっとも重要な項目です．カバレッジの検証対象は，あくまでも設計者や検証エンジニアが設定するもので，その内容が真に必要な項目をすべてカバーしているかどうかはだれにもわかりません．ほんとうに「神のみぞ知る」領域なのです．しかし，そんなことを言っていては検証の達成度はわかりません．そのため，わかっている限りの達成目標と重要度を指定することで，「理想」にいかに近づいているかを相対的に評価することになります．

● バッチ・モードでマルチセッション実行

通常の検証では，シミュレーションの条件やテスト・パターンを変更しながら，複数のシミュレーション・セッションを実行するのが一般的です．このようなシミュレーションでは，シミュレーション・セッションごとにグラフィカル・ウィンドウを使ってデバッグを行うことはありません．複数回のシミュレーションを自動的に実行する「バッチ・シミュレーション」を行い，各シミュレーションで保存されたログ・ファイルを使ってデバッグします．次のステップでは，バッチ・シミュレーションによってランダム・シードを変更しながら複数回のシミュレーションを実行し，保存されたカバレッジ・ログ・ファイルを結合して，シミュレーション全体のカバレッジ向上を確認します．

バッチ・モードの実行には，Windowsのコマンド・プロンプトを用います．この際，シミュレータのインストール・ディレクトリが実行パスに指定されている必要があります．インストール・ガイドに従ってインストールされていれば，問題なく実行できるはずですが，正しく実行できない場合には環境変数PATHに「C:\QuestaSim_6.1d\win32」を指定してください(デフォルト・ディレクトリにインストールした場合)．また，QuestaないしはModel Simのほかのバージョンをインストールしている場合には，「バージョン6.1d」の実行パスが先頭に来ていることを確認してください．

サンプル・データの中に，runtest.batというバッチ・ファイルを準備しました．内容を確認してください(**リスト4**)．本ファイルの中では，コンパイルの実行後`FOR`ループで`-sv_seed`を変更しながらシミュレーションを実行しています．ここでの設定では，`-sv_seed`の値を，10から5刻みで40まで7セッション実行しています．run.doフ

リスト4　バッチ・ファイル

```
vlog *.sv

FOR /L %%i IN (10, 5, 40) DO (
   vsim -sv_seed %%i top < run.do
   move tmp.cov %%i.cov
   )
```

FORループ．iを10から5刻みで40まで増加
Quetsa実行ファイル
ログ・ファイル作成

リスト5　Questa実行ファイル

```
run -all
fcover save tmp.cov
```

カバレッジ・ログ・ファイル保存

ァイルはシミュレータのコマンドを記述したファイル(**リスト5**)になります．この中では，シミュレーションの全イベント実行と，fcover saveコマンドによってtmp.covというファイルにカバレッジの結果を保存しています．各シミュレーション・セッションの終了後に，tmp.covファイルを<シードの値>.covというファイルにリネームしています．

バッチ・モードの実行のため，コマンド・プロンプトを起動して，「C:\example」へ移動します．プロンプトから，次のコマンドを実行します．

```
C:\examples> runtest
```

すると，シミュレーションを実行しているメッセージが表示され，カバレッジを記録した七つのログ・ファイル(xx.cov)が生成されます．

● 複数セッションの結果を確認

七つのログ・ファイルを次のコマンドを使って結合します．

```
C:\example> vcover merge all.cov 10.cov 15.cov 20.
cov 25.cov 30.cov 35.cov 40.cov
```

vcover　mergeコマンドに対して最初の引き数に「出力ファイル名」，そのあとに入力ファイルを列挙します．また，コマンド・プロンプト上で，各セッションのログからレポート作成を行えます．

```
C:\example> vcover　report　<ログ・ファイル名>
```

ここで，all.covを指定すれば，全セッションを結合したレポートを，10.covなどを指定すれば，各セッションのレポートを確認できます．また，下記の-outputオプションでレポートをファイルに出力することもできます．

```
C:\example> vcover　report　-output　all_report.txt
all.cov
```

次にGUIを起動して確認します．シミュレータを起動してtopをローディングします．その後，初期化を実行します．

```
QuestaSim> vsim　top
VSIM>　run　0
```

Analysisウィンドウで，covergroupが表示されるようにします．次のコマンドを実行して一つのシミュレーション・セッションのログをローディングします．

```
VSIM> fcover　reload　10.cov
```

ここに，15.covの結果を追加してローディングします．

```
VSIM> fcover　reload　15.cov　-merge
```

-mergeオプションによって，ログ・ファイルを累積モードでローディングします．このコマンドを使うことで，任意の複数ログ・ファイルを結合することができます．最後に，結合しておいたログ・ファイルをローディングします．

```
VSIM> fcover　reload　all.cov
```

all.covには，10.covと15.covの情報が含まれているため，ここでは-mergeは必要ありません．全体の結果から，hit_onceについてもかなりカバーされていることがわかります．cache_missは累積結果で100%を超えていますが，同一アドレスに対する5回のキャッシュ・ミスは，1回のシミュレーションの中でカバーされなければなりません．このようなカバレッジについては，セッションごとの結果を用いて評価する必要があります．

以上の操作から，ある程度複雑な検証も，SystemVerilogの機能を用いることで容易に検証できることを実感していただけたかと思います．

● 次回はアサーションについて解説

今回のテストベンチの拡張では，シミュレーション結果をデバッグする方法について触れていません．ランダム・パターン生成とカバレッジ計測は十分な項目を検証したかという指標であり，回路が正しく動作しているかどうかを検証するには別の方法論が必要になります．次回は，ファンクション・デバッグを効率化するアサーション・ベースの検証手法について紹介します．

もりた・えいいち
メンター・グラフィックス・ジャパン(株)

<筆者プロフィール>
森田栄一．Questa(旧名V-System)にかかわり始めたころは20代だったのに，気づけば40代間近．HDL設計環境を見続けてきて感じるのは，その進化の遅さ．設計規模と速度は2けたも向上しているのに，検証手法はやっとSystemVerilogなどが登場してきたにすぎない．そんなに速くてもついていけないのかもしれないが…．